Type Derivations

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

Objectives

In order to express the meaning of a program, we need a formal language to capture these meanings. One way to express meaning is to say something about the *types* of the expressions. By the end of lecture, you should know

- what the word “semantics” means.
- how to structure a proof-tree
- how use the type rules to prove the type of an expression
- write your own type rule for an expression

Strongly vs. Weakly Typed

Strongly Typed Languages

- e.g., ML, Haskell, most of C++, C
- You have to declare all your types at the beginning
- No type checks during run-time (faster, safer code)

Weakly Typed Languages

- e.g., Perl, Lisp, some parts of Java
- Very flexible programming
- Types must be checked at run-time or Bad Things happen

Other advantages of strong typing...

Engineering

- A program that type-checks is likely to be correct.
- Types constrain the use of a function.

Safety

- Every try casting an integer into a pointer in C?

```
1 int i = 5;
2 *((int *)i) = 10;
```

- A type error in an untyped language can be very hard to detect.

Theory

- It is much easier to verify the operation of type-correct code.

To create a formal system, you must specify the following:

- A set of *symbols* or an *alphabet*.
- A definition of a *valid sentence*.
- A set of *transformation rules* to make new valid sentences out of old ones.
- A set of *initial valid sentences*.

You do **NOT** need:

- An *interpretation* of those symbols.
  They are highly recommended, but the formal system can exist and do its work without one.

Symbols S, $(,)$, Z, P, x, y.

Definition of a furbitz

- Z is a furbitz. x and y are variables of type furbitz.
- if x is a furbitz, then $S(x)$ is a furbitz.
- if x and y are furbitzi, then $P(x, y)$ is a furbitz.

Definition of the gloppit relation

- Z has the gloppit relation with Z.
- If x and y have the gloppit relation, then $S(x)$ and $S(y)$ have the gloppit relation.
- If α and β, then we can write $\alpha g \beta$.

True Sentences If $\alpha g \beta$, then also

- $P(S(\alpha), \beta) g P(\alpha, S(\beta))$, and $P(Z, \alpha) g \alpha$

Symbols S, $(,)$, Z, P, x, y.

Definition of an integer

- 0 is an integer. x and y are variables of type integer.
- if x is an integer, then $S(x)$ is an integer.
- if x and y are integers, then $P(x, y)$ is an integer.

Definition of the equality relation

- 0 has the equality relation with 0.
- If x and y have the equality relation, then $S(x)$ and $S(y)$ have the equality relation.
- If α and β, then we can write $\alpha = \beta$.

True Sentences If $\alpha = \beta$, then also

- $P(S(\alpha), \beta) = P(\alpha, S(\beta))$, and $P(0, \alpha) = \alpha$

An *type judgment* has the following form:

$$\Gamma \vdash e : \tau$$

where Γ is a *type environment*, *e* is some expression, and τ is a *type*.

- $\Gamma \vdash$ `if true then` 4 `else` $38 :$ `int`
- $\Gamma \vdash$ `true && false` $:$ `bool`

Note: the $\vdash$ is pronounced “turnstile” or “entails”.

The Parts of a Rule

Assumptions

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

Conclusion

- If a rule has no assumptions, then it is called an *axiom*.
- Γ is a list of the form $[x : \tau; \ldots]$.
- Γ may be left out if we don't need a type environment (see next slide).
- **Basic Idea**: The meaning of an expression can be determined by combining the meaning of its parts.

Axioms

Constants $\dfrac{}{\vdash n : \texttt{int}}$ (assuming n is an int)

$$\frac{}{\vdash \texttt{true} : \texttt{bool}}$$

$$\frac{}{\vdash \texttt{false} : \texttt{bool}}$$

Variables $\dfrac{}{\Gamma \vdash x : \tau}$, if $x : \tau \in \Gamma$

- The Variable Rule is actually a bit more complicated in real life, but this form is sufficient.
- These are rules that are true no matter what the context is.

Simple Rules

Arithmetic

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{int}}$$

Relations

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \sim e_2 : \texttt{bool}}$$

Booleans

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{\&\&}\ e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{||}\ e_2 : \texttt{bool}}$$

- These are combination rules.

Simple Example

Suppose we want to prove that $\Gamma \vdash (x * 5 > 7)\texttt{\&\&}\ y : \texttt{bool}$.
Assume that $\Gamma = [x : \texttt{int}\ ; y : \texttt{bool}\ ]$

First thing: Write down the thing you are trying to prove, and put a bar over it.

$$\frac{}{\Gamma \vdash (x * 5 > 7)\texttt{\&\&}\ y : \texttt{bool}}$$

Look at the *outermost* expression. What kind of expression is this?

Use the rule $\dfrac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{\&\&}\ e_2 : \texttt{bool}}$

You need the "and" rule. It will tell you which parts of the goal need to be proved next.

Suppose we want to prove that $\Gamma \vdash (x * 5 > 7)\texttt{\&\&}\ y : \texttt{bool}$.
Assume that $\Gamma = [x : \texttt{int}\ ; y : \texttt{bool}\ ]$

Write parts on top and put a bar over them as well.

$$\frac{\overline{\Gamma \vdash x * 5 > 7 : \texttt{bool}} \qquad \overline{\Gamma \vdash y : \texttt{bool}}}{\Gamma \vdash (x * 5 > 7)\texttt{\&\&}\ y : \texttt{bool}}$$

What to do next? Let's work left to right. The expression we want next is a "greater" expression. (Besides, the y expression is already an axiom.)

Suppose we want to prove that $\Gamma \vdash (x * 5 > 7)\texttt{\&\&}\ y : \texttt{bool}$.
Assume that $\Gamma = [x : \texttt{int}\ ; y : \texttt{bool}\ ]$

Following the "greater" rule, we break the `x * 5 > 7` into two parts.

$$\frac{\dfrac{\overline{\Gamma \vdash x * 5 : \texttt{int}} \qquad \overline{\Gamma \vdash 7 : \texttt{int}}}{\Gamma \vdash x * 5 > 7 : \texttt{bool}} \qquad \overline{\Gamma \vdash y : \texttt{bool}}}{\Gamma \vdash (x * 5 > 7)\texttt{\&\&}\ y : \texttt{bool}}$$

We will turn our attention to the multiplication now.

Suppose we want to prove that $\Gamma \vdash (x * 5 > 7)\texttt{\&\&}\ y : \texttt{bool}$.
Assume that $\Gamma = [x : \texttt{int}\ ; y : \texttt{bool}\ ]$

At this point, there are no more subtrees to expand out. We are done.

$$\frac{\dfrac{\dfrac{\overline{\Gamma \vdash x : \texttt{int}} \qquad \overline{\Gamma \vdash 5 : \texttt{int}}}{\Gamma \vdash x * 5 : \texttt{int}} \qquad \overline{\Gamma \vdash 7 : \texttt{int}}}{\Gamma \vdash x * 5 > 7 : \texttt{bool}} \qquad \overline{\Gamma \vdash y : \texttt{bool}}}{\Gamma \vdash (x * 5 > 7)\texttt{\&\&}\ y : \texttt{bool}}$$

If $\dfrac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$

- The τ means "any type at all"—but whatever type τ you pick it has to be the same for the three places it shows up in this rule.
- So... the `if` rule says that `if` can result in any type, as long as the `then` and `else` branches have the same type. This could even include functions.

$$\frac{\Gamma \vdash e : \tau_1 \to \tau_2 \to \cdots \to \tau_n \to \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e\ e_1\ e_2\ \cdots\ e_n : \tau}$$

- If you have a function of type $\tau_1 \to \tau_2 \to \cdots \to \tau_n \to \tau$, and if every argument e_i has type τ_i, then applying them **in that order** will produce an expression of type τ.

$$\frac{\Gamma \vdash \texttt{map} : (\alpha \to \beta) \to \alpha\ \texttt{list} \to \beta\ \texttt{list} \quad \Gamma \vdash \texttt{f} : (\alpha \to \beta) \quad \Gamma \vdash \texttt{lst} : \alpha\ \texttt{list}}{\Gamma \vdash \texttt{map f lst} : \beta\ \texttt{list}}$$

- For "compound types" like α `list`, we only substitute in the α parts.

- Important point: the rules describe types, but they also describe when you may change Γ.
- You may **NOT** change Γ except as described!

Functions $$\frac{\Gamma \cup [\texttt{x}_1 : \tau_1; \cdots; \texttt{x}_n : \tau_n] \vdash e : \tau}{\Gamma \vdash \texttt{fun x}_1 \cdots \texttt{x}_n \texttt{->} e : \tau_1 \to \cdots \to \tau_n \to \tau}$$

Let $$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \cup [x : \tau] \vdash e_2 : \tau'}{\Gamma \vdash \texttt{let x} = e_1 \texttt{ in } e_2 : \tau'}$$

Let Rec $$\frac{\Gamma \cup [x : \tau] \vdash e_1 : \tau \quad \Gamma \cup [x : \tau] \vdash e_2 : \tau'}{\Gamma \vdash \texttt{let rec x} = e_1 \texttt{ in } e_2 : \tau'}$$

Type-checking compose:

$$\overline{\Gamma \vdash \texttt{fun f g x -> f (g x)} : (\alpha \to \beta) \to (\delta \to \alpha) \to \delta \to \beta}$$

Before using the function rule:

$$\frac{\Gamma \cup [\texttt{x}_1 : \tau_1; \cdots; \texttt{x}_n : \tau_n] \vdash e : \tau}{\Gamma \vdash \texttt{fun x}_1 \cdots \texttt{x}_n \texttt{->} e : \tau_1 \to \cdots \to \tau_n \to \tau}$$

Type-checking compose:

$$\frac{\overline{\Gamma' = \Gamma \cup \{\texttt{f} : (\alpha \to \beta); \texttt{g} : (\delta \to \alpha); \texttt{x} : \delta\} \vdash \texttt{f (g x)} : \beta}}{\Gamma \vdash \texttt{fun f g x -> f (g x)} : (\alpha \to \beta) \to (\delta \to \alpha) \to \delta \to \beta}$$

After using the function rule:

$$\frac{\Gamma \cup [\texttt{x}_1 : \tau_1; \cdots; \texttt{x}_n : \tau_n] \vdash e : \tau}{\Gamma \vdash \texttt{fun x}_1 \cdots \texttt{x}_n \texttt{->} e : \tau_1 \to \cdots \to \tau_n \to \tau}$$

Next, we use application.

$$\frac{\Gamma \vdash e : \tau_1 \to \tau_2 \to \cdots \to \tau_n \to \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e\ e_1\ e_2\ \cdots\ e_n : \tau}$$

Type-checking compose:

$$\dfrac{\dfrac{\overline{\Gamma' \vdash \texttt{f} \ : (\alpha \to \beta)} \qquad \overline{\Gamma' \vdash (\texttt{g}\ \texttt{x}) \ : (\alpha)}}{\Gamma' = \Gamma \cup \{\texttt{f} : (\alpha \to \beta); \texttt{g} : (\delta \to \alpha); \texttt{x} : \delta\} \vdash \texttt{f}\ (\texttt{g}\ \texttt{x}) \ : \ \beta}}{\Gamma \vdash \texttt{fun f g x -> f (g x)} : (\alpha \to \beta) \to (\delta \to \alpha) \to \delta \to \beta}$$

The rule for parens is trivial... can you write it?
Next, we'll use application again.

Type-checking compose:

$$\dfrac{\dfrac{\overline{\Gamma' \vdash \texttt{f} \ : (\alpha \to \beta)} \qquad \dfrac{\overline{\Gamma' \vdash \texttt{g} : (\delta \to \alpha)} \qquad \overline{\Gamma' \vdash \texttt{x} : \delta}}{\Gamma' \vdash (\texttt{g}\ \texttt{x}) \ : \alpha}}{\Gamma' = \Gamma \cup \{\texttt{f} : (\alpha \to \beta); \texttt{g} : (\delta \to \alpha); \texttt{x} : \delta\} \vdash \texttt{f}\ (\texttt{g}\ \texttt{x}) \ : \ \beta}}{\Gamma \vdash \texttt{fun f g x -> f (g x)} : (\alpha \to \beta) \to (\delta \to \alpha) \to \delta \to \beta}$$

We are done. These rules are meant mainly to *verify* a type, but they can be used to *infer* a type as well.

At first, give everything a separate type.

$$\overline{\Gamma \vdash \texttt{fun f g x -> f (g x)} : \alpha \to \beta \to \delta \to \gamma}$$

Apply the function rule:

$$\dfrac{\overline{\Gamma' = \Gamma \cup \{\texttt{f} : \alpha;\ \texttt{g} : \beta;\ \texttt{x} : \delta\} \vdash \texttt{f}\ (\texttt{g}\ \texttt{x}) \ : \ \gamma}}{\Gamma \vdash \texttt{fun f g x -> f (g x)} : \alpha \to \beta \to \delta \to \gamma}$$

Apply the application rule:

$$\frac{\dfrac{\overline{\Gamma' \vdash \texttt{f} \ : \alpha} \qquad \overline{\Gamma' \vdash (\texttt{g x}) \ :???}}{\Gamma' = \Gamma \cup \{\texttt{f} : \alpha;\ \texttt{g} : \beta;\ \texttt{x} : \delta\} \vdash \texttt{f}\ (\texttt{g x}) \ : \ \gamma}}{\Gamma \vdash \texttt{fun f g x} \texttt{ -> } \texttt{f}\ (\texttt{g x}) : \alpha \to \beta \to \delta \to \gamma}$$

From here we see that α needs to be a function type. Also, we need to decide a type for `(g x)`.

- Let `(g x)` have type ν.
- Let `f` have type $\alpha = \nu \to \gamma$.

We make the appropriate substitutions....

Apply the application rule:

$$\frac{\dfrac{\overline{\Gamma' \vdash \texttt{f} \ : (\nu \to \gamma)} \qquad \overline{\Gamma' \vdash (\texttt{g x}) \ : \nu}}{\Gamma' = \Gamma \cup \{\texttt{f} : (\nu \to \gamma);\ \texttt{g} : \beta;\ \texttt{x} : \delta\} \vdash \texttt{f}\ (\texttt{g x}) \ : \ \gamma}}{\Gamma \vdash \texttt{fun f g x} \texttt{ -> } \texttt{f}\ (\texttt{g x}) : (\nu \to \gamma) \to \beta \to \delta \to \gamma}$$

Now we use the application rule for `(g x)`.

Apply the application rule:

$$\frac{\dfrac{\overline{\Gamma' \vdash \texttt{f} \ : (\nu \to \gamma)} \qquad \dfrac{\overline{\Gamma' \vdash \texttt{g} : \beta} \qquad \overline{\Gamma' \vdash \texttt{x} : \delta}}{\Gamma' \vdash (\texttt{g x}) \ : \nu}}{\Gamma' = \Gamma \cup \{\texttt{f} : (\nu \to \gamma);\ \texttt{g} : \beta;\ \texttt{x} : \delta\} \vdash \texttt{f}\ (\texttt{g x}) \ : \ \gamma}}{\Gamma \vdash \texttt{fun f g x} \texttt{ -> } \texttt{f}\ (\texttt{g x}) : (\nu \to \gamma) \to \beta \to \delta \to \gamma}$$

- We know that β needs to be $\delta \to \nu$, because of our rule.

Substituting, we get...

Apply the application rule:

$$\frac{\dfrac{\overline{\Gamma' \vdash \texttt{f} \ : (\nu \to \gamma)} \qquad \dfrac{\overline{\Gamma' \vdash \texttt{g} : (\delta \to \nu)} \qquad \overline{\Gamma' \vdash \texttt{x} : \delta}}{\Gamma' \vdash (\texttt{g x}) \ : \nu}}{\Gamma' = \Gamma \cup \{\texttt{f} : (\nu \to \gamma);\ \texttt{g} : (\delta \to \nu);\ \texttt{x} : \delta\} \vdash \texttt{f}\ (\texttt{g x}) \ : \ \gamma}}{\Gamma \vdash \texttt{fun f g x} \texttt{ -> } \texttt{f}\ (\texttt{g x}) : (\nu \to \gamma) \to (\delta \to \nu) \to \delta \to \gamma}$$

Now we're done.

Here's an example I showed one time I gave this lecture.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \cup [\texttt{x} : \tau] \vdash \texttt{x} : \tau}}{\Gamma \vdash \texttt{fun x ->x} : \tau \to \tau} \quad \dfrac{\dfrac{}{\Gamma^1 \vdash \texttt{id} : \tau \to \tau} \quad \dfrac{\dfrac{}{\Gamma^1 \vdash \texttt{id} : \tau \to \tau} \quad \dfrac{}{\Gamma^1 \vdash 10 : \texttt{in}}}{\Gamma^1 \vdash (\texttt{id}\ 10) : \texttt{int}}}{\Gamma^1 \vdash \texttt{id}\ (\texttt{id}\ 10) : \texttt{int}}}{\Gamma \vdash \texttt{let id} = \texttt{fun x ->x in id}\ (\texttt{id}\ 10) : \texttt{int}}$$

- Let $\Gamma^1 = \Gamma \cup [\texttt{id} : \tau \to \tau]$
- What would be different if we'd used `let rec` instead?

Try these problems.

- Prove that $\Gamma \vdash \texttt{let}\ f = \texttt{fun}\ x \texttt{->} x + 2\ \texttt{in let}\ g = \texttt{fun}\ x \texttt{->} x + 3\ \texttt{in}\ (\texttt{if}\ 4 > 6\ \texttt{then}\ f\ \texttt{else}\ g)\ 10 : \texttt{int}$
- Write a type judgment rule for the list operator `::`.